SANDIA REPORT

SAND2007-5984
Unlimited Release
Printed August 2010

Thyra Linear Operators and Vectors

Overview of Interfaces and Support Software for the Development and Interoperability of Abstract Numerical Algorithms

Roscoe A. Bartlett
Optimization and Uncertainty Quantification

Sandia National Laboratories

SAND2007-5984
Unlimited Release
Printed August 2010

Thyra Linear Operators and Vectors

Overview of Interfaces and Support Software for the Development and Interoperability of Abstract Numerical Algorithms

Roscoe A. Bartlett
Optimization/Uncertainty Estim

Abstract

Engineering and scientific applications are becoming increasingly modular, utilizing publicly defined interfaces to integrate third party tools and libraries for services such as mesh generation, data partitioning, equation solvers and optimization. As a result, it is important to understand and model the interactions between these various modules, and to develop good abstract interfaces between them. One category of modules that is becoming increasingly important is abstract numerical algorithms (ANAs). ANAs such as linear and nonlinear equation solvers, methods for stability and bifurcation analysis, uncertainty quantification methods and nonlinear programming solvers for optimization are typically mathematically sophisticated but have surprisingly little essential dependence on the details of what computer system is being used or how matrices and vectors are stored and computed. As a result, using abstract interface capabilities in languages such as C++, we can implement ANA software that it will work, unchanged, with a variety of applications and linear algebra libraries.

In this paper, we provide an overview of the Thyra effort which at its most basic level defines fundamental abstract linear operator and vector interfaces. These linear operator/vector interfaces provide the basic functionality and interoperability for a broad range of ANAs. Many other higher-level abstractions are built on top of the Thyra operator/vector interfaces. The Trilinos package `Thyra` defines these different sets of C++ interfaces and provides optional support software.

Acknowledgment

The authors would like to thank everyone on the Trilinos team for their support in the Thyra effort.

The format of this report is based on information found in [15].

Contents

Figures

1 Introduction

One area of steady improvement in large-scale engineering and scientific applications is the increased modularity of application design and development. Specification of publicly-defined interfaces, combined with the use of third-party software to satisfy critical technology needs in areas such as mesh generation, data partitioning and solution methods have been generally positive developments in application design. While the use of third party software introduces dependencies from the application developer's perspective, it also gives the application access to the latest technology in these areas, amortizes library and tool development across multiple applications and, if properly designed, gives the application easy access to more than one option for each critical technology area, e.g., access to multiple linear solver packages.

One category of modules that is becoming increasingly important is abstract numerical algorithms (ANAs). ANAs such as linear and nonlinear equation solvers, methods for stability and bifurcation analysis, transient solvers, uncertainty quantification methods, and nonlinear programming solvers for optimization are typically mathematically sophisticated but have surprisingly little essential dependence on the details of what computer system is being used or how matrices and vectors are stored and computed. Thus, by using abstract interface capabilities in languages such as C++, we can implement ANA software such that it will work, unchanged, with a variety of applications and linear algebra libraries.

Here we describe a set of abstract operator/vector interfaces that allows the specification of ANAs from basic Krylov linear equation solvers all the way up to interior-point methods for optimization. At the core, we define a set of basic operator/vector interfaces that form the the foundation for (i) ANA development, (ii) the integration of an ANA into an application (APP) and (iii) providing services to the ANA from a linear algebra library (LAL). By agreeing on a simple minimal common interface layer such as the Thyra Fundamental ANA Operator/Vector Interfaces described here, we eliminate the many-to-many dependency problem of ANA/APP interfaces.

It is difficult to describe a set of linear algebra interfaces outside of the context of some class of numerical problems. For this purpose, we will consider numerical algorithms where it is possible to implement all of the required operations exclusively through well defined interfaces to vectors, vector spaces, and linear operators and higher level abstractions built on these. The fundamental Thyra operator/vector interfaces described here are the common denominator of all abstract numerical algorithms.

We assume that the reader has a basic understanding of vector reduction/transformation operators (RTOp) [3], is comfortable with object-orientation [11] and C++, and knows how to read basic Unified Modeling Language (UML) [10] class diagrams. We also assume that the reader has some background in large-scale numerics and will therefore be able to appreciate the challenges that are addressed by Thyra.

Note that the online documentation for Thyra at

```
http://trilinos.sandia.gov/packages/thyra
```

should be the definitive information source for Thyra. This document only tries to provide an overview of Thyra and explain the philosophy behind it.

2 Classification of linear algebra and other interfaces

Although we will discuss APPs, ANAs and LALs in detail later in this section, we want to briefly introduce these terms here to make them clear. Also, although there are certainly other types of modules in a large-scale scientific/engineering application, we only focus on these three since they are the ones more directly related to ANAs.

- Application (APP): The modules of an application that are not ANA or LAL modules. Typically this includes the code that is unique to the application itself such as the code that formulates and generates the discrete problem to be solved. In general it would also include other third-party software that is not an ANA or LAL module.
- Abstract Numerical Algorithm (ANA): Software that drives a solution process, e.g., an iterative linear or nonlinear solver. This type of package provides solutions to and requires services from the APP, and utilizes services from one or more LALs. It can usually be written so that it does not depend on the details of the computer platform, or the details of how the APP and LALs are implemented, so that an ANA can be used across many APPs and with many LALs.
- Linear Algebra Library (LAL): Software that provides the ability to construct concrete linear algebra objects such as matrices and vectors. A LAL can also be a specific linear solver or preconditioner.

An important focus of this paper is to clearly identify the interactions between APPs, ANAs and LALs for the purposes of defining the Thyra interfaces and to differentiate the Thyra interfaces from other interfacing efforts.

The requirements for the linear algebra objects as imposed by an ANA are very different from the requirements imposed by an APP code. In order to differentiate the various types of interfaces and the requirements associated with each, consider Figure 1. This figure shows the three major categories of software modules that make up a complete numerical application. The first category is application (APP) software in which the underlying data is defined for the problem. This could be something as simple as the right-hand-side and matrix coefficients of a single linear system or as complex as a finite-element method for a 3-D nonlinear PDE-constrained optimization problem. The second category is linear algebra library (LAL) software that implements basic linear algebra operations [9, 1, 5, 13, 2, 12]. These types of software include primarily the implementations for matrix-vector multiplication, the creation of a preconditioner (e.g. ILU), and may even include several different types of direct linear solvers. The third category is ANA software that drives the main high-level solution process and includes such algorithms as iterative methods for linear and nonlinear systems; explicit and implicit methods for ODEs and DAEs; and nonlinear programming (NLP) solvers [16]. There are many example software packages [2, 13, 12, 7, 4] that contain ANA software.

The types of ANAs described here only require operations like matrix-vector multiplication, linear solves and certain types of vector reduction and transformation operations. All of these operations can be performed with only a very abstract view of vectors, vector spaces and linear operators.

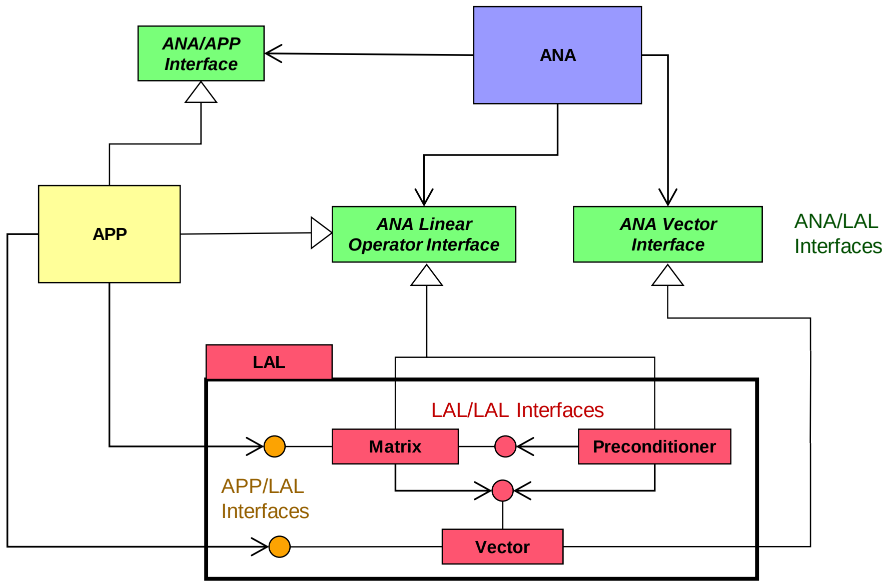

Figure 1. UML [6] class diagram : Interfaces between abstract numerical algorithm (ANA), linear algebra library (LAL), and application (APP) software.

An application code, however, has the responsibility of populating vector and matrix objects and requires the passing of explicit function and gradient value entries, sometimes in a distributed memory parallel environment. This is the purpose of an APP/LAL interface. This involves a very different set of requirements than those described above for the ANA/APP and ANA/LAL interfaces. Examples of APP/LAL interfaces include the FEI [8] and much of the ESI [14].

Figure 1 also shows a set of LAL/LAL interfaces that allows linear algebra objects from one LAL to collaborate with the objects from another LAL. Theses interfaces are very similar to the APP/LAL interfaces and the requirements for this type of interface is also not addressed by Thyra. The ESI [14] contains examples of LAL/LAL interfaces.

The primary distinction between ANA (ANA/APP and ANA/LAL) and LAL (APP/LAL and LAL/LAL) interfaces is that large amounts of concrete data are passed through LAL interfaces but very little data is passed through ANA interfaces. For example, the Thyra fundamental ANA operator/vector interfaces only pass very small arrays or single values of scalar floating point and integral datatypes. On the other hand, LAL interfaces for vector and matrix coefficient loading and accessing involve the direct manipulation and passing of large amounts of floating-point and integral data objects. Because of the large amount of explicit data passed to and from the LAL interfaces, they must have explicit knowledge of the computing platform, the distribution of data to in various memory levels, and other complex details. Getting near optimal performance out of LAL interfaces may require specialized application-specific and/or linear algebra implementation specific interfaces. Because of this, LAL interfaces are necessarily much more complex and less general than ANA interfaces and present challenges to the usage of runtime polymorphism (where non-inlined virtual function call overhead may be large). Alternatively, ANA interfaces typically hide large amounts of work per function call and pass very little data and therefore are well suited to runtime polymorphic interfaces (e.g. using abstract base classes in C++ using virtual functions).

3 Some Basic Requirements for Thyra

Before describing the C++ interfaces for Thyra, some basic requirements are stated.

1. Thyra interfaces should be portable to all reasonable ASC [17] and office of science platforms and environments where applications might run in addition to other important platforms.
2. Thyra interfaces should provide for stable and accurate numerical computations at a fundamental level.
3. Thyra should provide a minimal, but comprehensive, interface that addresses all the basic efficiency needs (in both speed and storage) which will result in near-optimal implementations of all of the objects and all of the above mentioned ANA algorithms that use these objects.
4. Thyra interfaces should efficiently and maintainably facilitate the development of basic compositional design patterns [11] such as DECORATOR and COMPOSITE.
5. The work required to implement adapter subclasses (see the "Adapter" pattern in [11]) for and with Thyra should be minimal and straightforward for all of the existing related linear algebra and ANA interfaces. This requirement is facilitated by the fact that the Thyra interfaces are minimal.
6. Maximally general ANAs developed with Thyra should be able to transparently utilize different types of computing environments such as SPMD[1], client/server[2], out-of-core[3], complex multi-core architectures[4] and any arbitrary combination of these configurations.

A hand-coded program (e.g. using Fortran 77 and MPI) should not provide any significant gains in performance in any of the above categories in any computing environment or configuration. A hand-coded algorithm in Fortran 77 with MPI should not be able to provide significant improvements in storage requirements, computational speed, or numerical stability. There are many numerical algorithms cannot be considered to be "abstract" (e.g. Gaussian Elimination) and therefore Thyra and like abstract interfaces should not be used for such algorithms. However, drawling the line between an ANA and a non-ANA can be quite fuzzy in practice.

[1]Single Program Multiple Data (SPMD): A single program running in a distributed-memory environment on multiple parallel processors

[2]Client/Server: The ANA runs in a process on a client computer and the APP and LAL run in processors on a server

[3]Out-of-core: The data for the problem is stored on disk and is read from and written to back disk as needed

[4]Multi-core architectures: Complex threading models and GPU platforms.

4 Mathematical Foundation for Fundamental ANA Operator/Vector Interfaces

Before describing the *Thyra Fundamental ANA Operator/Vector Interfaces* that form the foundation for all of the Thyra ANA interfaces in more detail, we must first clearly define the mathematical formulations suppored by Thyra for the abstractions of vectors, vector spaces, and linear operators.

All Thyra vectors belong to a vector space and are representable as a finite-dimensional array of scalar complex-valuded (or real valued) coefficients $\tilde{x} \in \mathbb{C}^n$ and a (non-unique) basis $E \in \mathbb{R}^{n \times n}$ of the form

$$x = E\tilde{x}. \tag{1}$$

For a given Thyra vector space, the basis representation $E \in \mathbb{C}^{n \times n}$ is non-unique but the symmetric positive definite scalar (inner) product matrix

$$Q = E^T E \tag{2}$$

is unique where $Q \in \mathbb{R}^{n \times n}$. Given Q, the scalar product is defined as

$$x^H y = \tilde{x}^H Q \tilde{y} = < \tilde{x}, \tilde{y} > . \tag{3}$$

For Q to be full rank, E must be full rank. Here we define a Euclidean vector space as one where $E = I$ and $Q = E^H E = I$.

In Thyra, vector spaces and vectors are abstracted using the C++ base interface classes `Thyra::VectorSpaceBase` and `Thyra::VectorBase` respectively. Vectors are created from a vector space using the ABSTRACT FACTORY design pattern using the nonmember function `vec=createMember(vecSpc)`.

In addition to vectors and vector spaces, Thyra also defines linear operators which map vectors from one vector space to another. The definition of a linear operator is strongly influenced by the definition of the scalar product associated with the vector spaces and whether the mapping is between Euclidean vectors $y = Ax$ or between coefficient vectors $\tilde{y} = \tilde{A}\tilde{x}$. In Thyra, vectors are a specialization of linear operators and therefore every vector is also a linear operator. Therefore, when one writes

$$z = x^H y \tag{4}$$

then x^H must be interpreted to be the adjoint linear operator of x.

Given this notation, the vector x would be considered to live in a Euclidean vector space $x \in \mathbb{C}^n$ while the vector $\tilde{x}$ would be considered to live in the non-Euclidean vector space $\tilde{x} \in \mathcal{X}$ which is defined by the scalar product matrix $Q \in \mathbb{R}^{n \times n}$. Therefore, it is actually not clear whether an abstract vector object represents the Euclidean vector $x \in \mathbb{C}^n$ which just happens to be stored as a set of coefficients $\tilde{x} \in \mathcal{X}$ or if it represents the coefficient vectors $\tilde{x} \in \mathcal{X}$ themselves.

5 Overview of Fundamental Thyra ANA Operator/Vector Interfaces

The Fundamental Thyra ANA Operator/Vector Interfaces are shown in Figure 2. The key abstractions include vectors, vector spaces, and linear operators. All of the interfaces are templated on a `Scalar` type[5] but the UML notation for templated classes is not used in the figure for the sake of improving readability. The memory management in Thyra is based on the Teuchos memory management classes [**?**] which means it involves no raw pointers and largely eliminates undefined behavior that is otherwise typical in C++ development.

Vector space is the foundation for all other abstractions. Vector spaces are abstracted through the *`VectorSpaceBase`* interface. A *`VectorSpaceBase`* object acts primarily as an ABSTRACT FACTORY [11] that creates vector objects (which are the "products" in the design pattern).

Vectors are abstracted through the *`VectorBase`* interface. The *`VectorBase`* interface is very minimal and really only defines one nontrivial function *`applyOp(...)`*. The *`applyOp(...)`* function accepts user-defined (i.e. ANA-defined) reduction/transformation operator (RTOp) objects through the templated RTOp C++ interface *`RTOpPack::RTOpT`*. An ever increasing set of concrete implementations of RTOps is provided along with wrapper convenience functions in the Thyra ANA support code collection. The set of RTOp element-wise reduction/transformation operations is also easily extensible and does not require any changes at all to any existing vector implementation (which is the whole point). Every *`VectorBase`* object provides access to its *`VectorSpaceBase`* (that was used to create the *`VectorBase`* object) through the function `space()`.

2011/09/16: ToDo: Finished editing up to here!

The *`VectorSpaceBase`* interface also provides the ability to create *`MultiVectorBase`* objects through the *`createMembers(numMembers)`* function. A *`MultiVectorBase`* is a tall thin dense matrix where each column in the matrix is a *`VectorBase`* object which is accessible through the *`col(...)`* function. *`MultiVectorBase`*s are needed for near-optimal processor cache performance (in serial and parallel programs) and to minimize the number of global communications in a distributed parallel environment. The *`MultiVectorBase`* abstraction is critical in many different types ANAs such as block Krylov methods. The interface class *`VectorBase`* is derived from *`MultiVectorBase`* so that every *`VectorBase`* object is also a *`MultiVectorBase`* object as well. This simplifies the development of ANAs in that any ANA that can handle *`MultiVectorBase`* objects can also automatically accept *`VectorBase`* objects as well. (However, the derivation of the vector interface from the multi-vector interface does lead to some increased complexity in creating more derived interface subclasses by requiring the use of multiple inheritance.)

`VectorSpaceBase` declares a virtual function called *`scalarProd(x,y)`* which computes the scalar product $< x, y >$ for the vector space. There is also a *`MultiVectorBase`* version

[5]`Scalar` types used with Thyra can be `double`, `float`, `complex<float>`, `complex<double>`, or any other concrete value-type that has a specialization of `Teuchos::ScalarTraits` which includes various extended precision types.

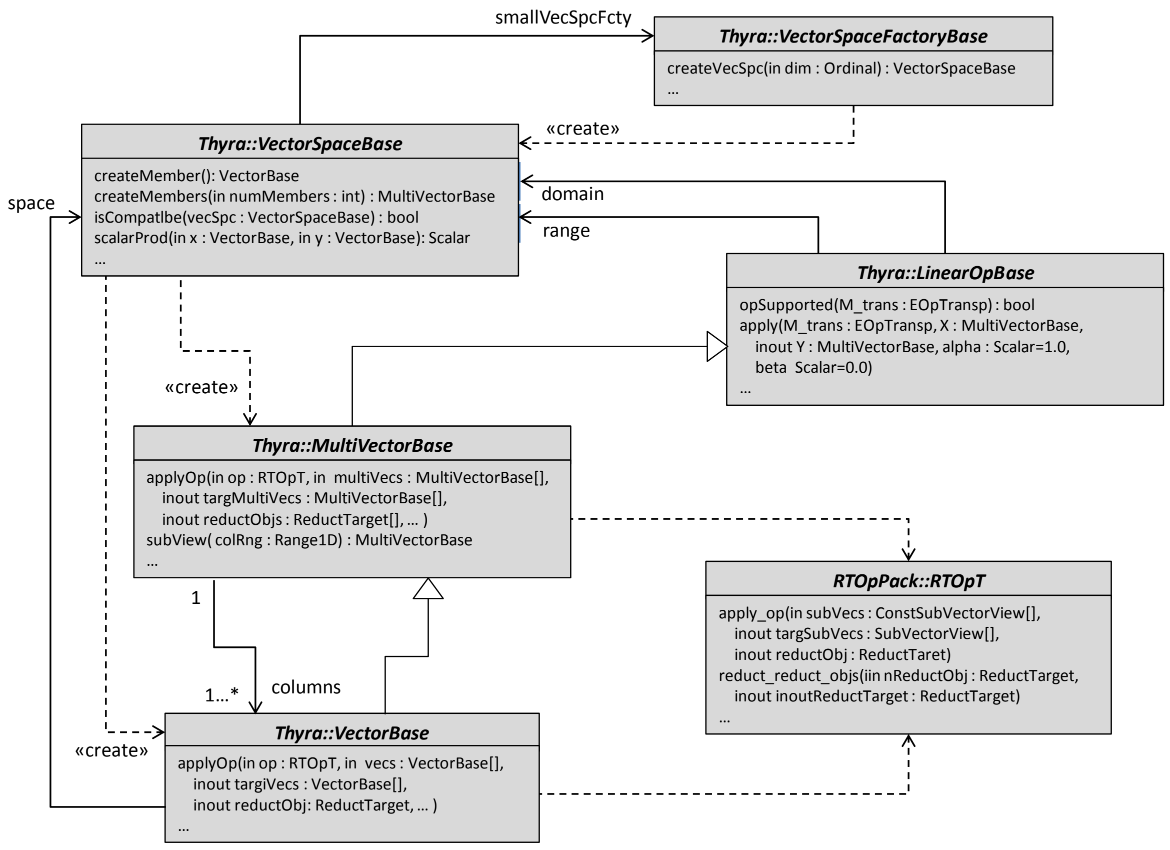
smallVecSpcFcty
Thyra::VectorSpaceFactoryBase
createVecSpc(in dim : Ordinal) : VectorSpaceBase
…
«create»
Thyra::VectorSpaceBase
createMember(): VectorBase
createMembers(in numMembers : int) : MultiVectorBase
isCompatlbe(vecSpc : VectorSpaceBase) : bool
scalarProd(in x : VectorBase, in y : VectorBase): Scalar
…
space
domain
range
Thyra::LinearOpBase
opSupported(M_trans : EOpTransp) : bool
apply(M_trans : EOpTransp, X : MultiVectorBase,
inout Y : MultiVectorBase, alpha : Scalar=1.0,
beta Scalar=0.0)
…
«create»
Thyra::MultiVectorBase
applyOp(in op : RTOpT, in multiVecs : MultiVectorBase[],
inout targMultiVecs : MultiVectorBase[],
inout reductObjs : ReductTarget[], …)
subView(colRng : Range1D) : MultiVectorBase
…
RTOpPack::RTOpT
apply_op(in subVecs : ConstSubVectorView[],
inout targSubVecs : SubVectorView[],
inout reductObj : ReductTaret)
reduct_reduct_objs(iin nReductObj : ReductTarget,
inout inoutReductTarget : ReductTarget)
…
1
1…*
columns
«create»
Thyra::VectorBase
applyOp(in op : RTOpT, in vecs : VectorBase[],
inout targiVecs : VectorBase[],
inout reductObj: ReductTarget, …)
…

Figure 2. UML class diagram : The Thyra fundamental ANA operator/vector interfaces

VectorSpaceBase::scalarProds(...) (not shown in the figure) that computes the scalar products of each set of column vectors in two multi-vectors. Finally, *VectorSpaceBase* includes the ability to determine the compatibility of (multi)vectors from different vector spaces through the function *isCompatible(vecSpc)*. This is useful primarily for error checking and debugging.

Another important type of linear algebra abstraction is a linear operator which is represented by the interface class *LinearOpBase*. The *LinearOpBase* interface is used to represent quantities such as a derivative operators for nonlinear vector-valued functions $f(x)$ such as the Jacobian matrix $\partial f / \partial x$. A *LinearOpBase* object defines a linear mapping for (multi)vectors in one vector space (called the `domain`) to (multi)vectors in another vector space (called the `range`). Every *LinearOpBase* object provides access to these vector spaces through the functions *domain()* and *range()*. The exact form of this mapping, as implemented by the function *apply(...)*, is

$$Y = \alpha\, op(M)\, X + \beta Y \tag{5}$$

where M is a *LinearOpBase* object; $op(M)$ is M or M^H; X and Y are *MultiVectorBase* objects; and α and β are `Scalar` objects.

The function *opSupported(M_trans)* will return `false` if a particular form of the transpose is not supported. Specifically, if the adjoint apply is supported, then *opSupported(CONJTRANS)* returns `true`.

If the adjoint is supported, then it must satisfy the adjoint property Specifically, for any two vectors $w \in \mathcal{D}$ (domain space) and $u \in \mathcal{R}$ (range space), the adjoint operation must obey the adjoint property

$$< u, Av >_{\mathcal{R}} == < A^H u, v >_{\mathcal{D}} .$$

Another important part of this design is the fact that *MultiVectorBase* derives from *LinearOpBase* and therefore every multi-vector object is also a linear operator. This is an elegant way to support the notions of block inner products and block updates.

A multi-vector block inner product is specified as

$$Z = Y^H X$$

where Y, X and Z are all multi-vectors. Note that since Y is a linear operator then $Y^H X$ is not simply the block dot product involving the coefficients but instead must be consistent with the scalar product associated with the range space of Y.

A multi-vector block update takes the form

$$Z = \alpha Y X + \beta Z$$

where Y, X and Z are all multi-vectors and α and β are scalars.

Together with basic linear operators applications, block inner products and block updates form the basic operations needed to implement block Krylov and other block numerical aglorithms.

Also note that since *MultiVectorBase* derives from *LinearOpBase* and *VectorBase* derives from *MultiVectorBase*, therefore, every vector object is also a linear operator. While this may not be a terribly useful feature it does mean that one must interpret $y^H x$ to be the same as $< y, x >$ and not just the dot product when a non-Euclidean vector space is involved.

6 Interpretations and limitations of Thyra interfaces

The Thyra Fundamental ANA Operator/Vector Interfaces require that every Thyra vector be represented as a finite-dimensional set of scalar coefficients and that the scalar product $<\tilde{x},\tilde{y}>$ be equivalent to the two-sided application of some finite-dimensional symmetric positive definite matrix $Q \in \mathbb{R}^{n \times n}$ such that $<\tilde{x},\tilde{y}>=\tilde{x}^H Q \tilde{y}$. The Thyra interfaces do not try to pretend that it's vectors are infinite dimensional or that they admit more general implementations as allowed by infinite dimensional Hilbert spaces. While every Thyra vector must be stored as a set of scalar coefficients, the interfaces make no assumptions whatsoever about where or how it's coefficients are stored. A fully general ANA can make no assumptions about how vectors are stored or laid out in memory (or even if any of the elements are cheaply accepssible), only that those coefficients do exist and that the coefficients can be exposed to reduction and transformation operators (see [3]).

While accessing the elements of a vector is ill advised in a general ANA, the Thyra vector interface supports acquiring direct views of any range of vector coefficient data (see `Thyra::VectorBase::acquireDetachedView(...)`).

Again, Thyra interfaces do not try to hide the fact that a vector is simply a set of scalar coefficients associated with some basis. Thyra does not even really try to keep the client ANA from accessing the actual coefficients of the vectors since it can access them with an RTOp object if desired. What Thyra does do is to abstract where the vector coefficients live and what native data structure is used to hold the coefficients. None of vector coefficients may even be directly be head in main memory in the process where the ANA is running (e.g. GPUs) but there is aways a (perhaps very inefficient) mechanism to get a view of and retrieve them using RTOps. ANAs that want to be maximally general and efficient should not try to directly access the vector elements explicitly and many ANAs do not need to. However, there are perfectly reasonable ANAs that do need to access the explicit vector coefficients for vectors from certain vector spaces (such as vectors in the design space in some reduced-space optimization algorithms in a solver such as MOOCHO [?]) and Thyra provides efficient and yet 100% general ways to access these coefficients. Every Thyra vector space also has a finite dimension that is accessible as a property of the vector space and is exposed as a (64 bit) integral datatype through `VectorSpaceBase::dim()`.

While Thyra requires vectors to be finite dimensional and the vector coefficients must be accessible (if to none other than to RTOps), it allows complete freedom in the implementation of scalar products and of general linear operators that map vectors from one vector space to another.

7 Summary

The Thyra Fundamental Operator/Vector Interfaces provide the intersection of a comprehensive range of the functionality required of linear operators and vectors by a variety of abstract numerical algorithms ranging from basic iterative linear solvers all the way up to interior-point methods for optimization. By adopting Thyra as a standard interface layer, interoperability between applications, linear algebra libraries, and abstract numerical algorithms in advanced scientific computing environments becomes automatic to a large extent.

References

[1] E. Anderson et al. *LAPACK User's Guide.* SIAM, 1995.

[2] S. Balay, W. D. Gropp, L.C. McInnes, and B.F. Smith. PETSc 2.0. http://www.mcs.anl.gov/petsc.

[3] R. A. Bartlett, B. G. van Bloeman Waanders, and M. A. Heroux. Vector reduction/transformation operators for linear algebra interfaces to efficiently develop complex abstract numerical algorithms independently of data mapping, 2003. Submitted to *ACM TOMS*.

[4] Steve Benson, Lois Curfman McInnes, and Jorge Moré. TAO : Toolkit for advanced optimization (web page).

[5] L. S. Blackford, J. Choi, A. Cleary, E.D. 'Azevedo, J. Demmel, I. Dhilon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walder, and R.C. Whaley. *ScalLAPACK User's Guide.* SIAM, Philadelphia, PA, 1997.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[7] George D. Byrne and Allan C. Hindmarsh. PVODE, an ODE solver for parallel computers. *Int. J. High Perf. Comput. Applic*, 13:354–365, 1999.

[8] Robert L. Clay, Kyran D. Mish, Ivan J. Otero, Lee M. Taylor, and Alan B. Williams. An annotated reference guide to the finite-element interface (FEI) specification : Version 1.0. Technical Report SAND99-8229, Sandia National Laboratories, 1999.

[9] J. Demmel. *Applied Numerical Linear Algebra.* SIAM, 1997.

[10] M. Fowler and K. Scott. *UML Distilled, second edition.* Addison-Wesley, 2000.

[11] E. Gamma et al. *Design Patterns: Elements fo Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[12] Mike A. Heroux, Teri Barth, David Day, Rob Hoekstra, Rich Lehoucq, Kevin Long, Roger Pawlowski, Ray Tuminaro, and Alan Williams. Trilinos : object-oriented, high-performance parallel solver libraries for the solution of large-scale complex multi-physics engineering and scientific applications. `http://software.sandia.gov/Trilinos`.

[13] S. A. Hutchinson, L. V. Prevost, J.N. Shadid, C. Tong, and R.S. Tuminaro. Aztec user's guide: Version 2.0. Technical Report ANL-95/11–Revision 2.0.22, Sandia National Laboratories, 1998.

[14] Sandia National Labs. ESI: Equation solver interface. `http://z.ca.sandia.gove/esi`, 2001.

[15] Tamara K. Locke. Guide to preparing SAND reports. Technical report SAND98-0730, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 1998.

[16] J. Nocedal and S. Wright. *Numerical Optimization*. Springer, New York, 1999.

[17] Department of Energy. ASCI: Advanced simulation and computing initiative. `http://www.sandia.gov/ASCI`.

DISTRIBUTION:

2	MS 9018	Central Technical Files, 8944
2	MS 0899	Technical Library, 4536

Sandia National Laboratories